

# Evolutionary Trends of Developer Coordination: A Network Approach

Mitchell Joblin · Sven Apel · Wolfgang Mauerer

Received: date / Accepted: date

**Abstract** Software evolution is a fundamental process that transcends the realm of technical artifacts and permeates the entire organizational structure of a software project. By means of a longitudinal empirical study of 18 large open-source projects, we examine and discuss the evolutionary principles that govern the coordination of developers. By applying a network-analytic approach, we found that the implicit and self-organizing structure of developer coordination is ubiquitously described by non-random organizational principles that defy conventional software-engineering wisdom. In particular, we found that: (a) developers form scale-free networks, in which the majority of coordination is primarily managed by an extremely small number of developers, (b) developers tend to become more coordinated over time, limited by an upper bound, and (c) initially developers are hierarchically arranged, but over time, form a unique hybrid structure, in which core developers are hierarchically arranged and peripheral developers are not. Our results suggest that the organizational structure of large projects is constrained to evolve towards a state that balances the costs and benefits of developer coordination, and the mechanisms used to achieve this state depend on the project's scale.

**Keywords** Software Evolution · Developer Coordination · Developer Networks

---

Mitchell Joblin  
Siemens AG  
Wladimirstrasse 3, 91058 Erlangen, Germany  
Tel.: +49-176-61335168  
E-mail: mitchell.joblin.ext@siemens.com

Sven Apel  
University of Passau  
Innstr. 33, 94032 Passau, Germany  
Tel.: +49-851-5093225  
E-mail: apel@uni-passau.de

Wolfgang Mauerer  
Technical University of Applied Science Regensburg  
Universitätsstrasse 31, 93058 Regensburg, Germany  
Tel.: +49-941-9439753  
E-mail: wolfgang.mauerer@oth-regensburg.de

## 1 Introduction

Change in software is inevitable, and the constant pressure to adapt is a challenge that all software projects encounter. The necessity of change is not isolated to the software design and implementation, it permeates through all artifacts and facets of a project including the entire *organizational structure*. As the software evolves, the organizational structure building the software must also evolve to maintain effective coordination between developers. In the ideal case, a match or congruence is achieved between the coordination needs implied by the project's technical artifact structure and the coordination mechanisms implied by the developer's organizational structure (Cataldo et al, 2008).

The need for developer coordination is largely a consequence of software-artifact interdependencies. For example, two developers independently constructing coupled components must coordinate to avoid violating assumptions embedded in the components' design. During software evolution, artifact interdependencies are added, removed, or changed, and local changes can propagate to dependent artifacts and alter the requirement for developers to coordinate. Without coherence between the artifact structure and the organizational structure, the developers may lose awareness of new dependencies and their effects (Cataldo et al, 2009; Sosa et al, 2004). Architectural documentation can help support dependency awareness, however, it is difficult to maintain accurate documentation for an evolving system, and certain interdependencies may not be obviously expressed in the source code (e.g., code clones). Empirical evidence has demonstrated that the loss of interdependency awareness negatively influences software quality and developer productivity (Espinosa et al, 2007; Cataldo et al, 2009; Cataldo and Herbsleb, 2013).

Our goal to understand the mechanisms that govern software evolution is motivated by specific scaling constraints known to affect software engineering. One such scaling constraint arises from the quadratic relationship between team size and the number of possible interactions between developers. For example, a group of  $n$  developers can each coordinate with  $n - 1$  other developers in the group such that the total number of possible coordination requirements is  $n(n - 1) = \mathcal{O}(n^2)$ . The implication is that, at a critical point, the overhead involved in coordination may exceed the benefit of having additional developers (Brooks, 1995). Consequently, the organizational structure of successful projects is constrained to evolve in a manner that mitigates the negative effects of scaling constraints. We expect that the influence of the scaling constraints will be observable through the evolutionary trends of the organizational structure.

Methodologically, we use a network-analytic framework to conduct an exploratory study on the evolution of developer organization with respect to the following three well-known and statistically well-founded organizational principles:

- **Scale freeness.** Scale-free networks are characterized by the existence of hub nodes with an extraordinarily large number of connections, which results in several beneficial characteristics including robustness and scalability (Dorogovtsev and Mendes, 2013). Developer networks of this kind are conjectured to tolerate substantial breakdowns in coordination without significant consequences to software quality (Dorogovtsev and Mendes, 2013; Cataldo and Herbsleb, 2013).
- **Modularity.** The local arrangement of nodes into groups that are internally well connected gives rise to a modular structure. Modularity is a notable characteristic of many complex systems and indicates the specialization of functional modules. In the case of developer organization, this is the primary mechanism used to reduce system-wide coordination overhead and increase productivity (Brooks, 1995).

- **Hierarchy.** The global arrangement of nodes into a layered structure, where small cohesive groups are embedded within larger and less cohesive groups, forms a hierarchy. Hierarchy is an organizational principle distinct from modularity and scale freeness, and has been shown to improve the coordination of distributed teams (Hinds and McGrath, 2006). For developer networks, hierarchy suggests the existence of stratification within the developer roles and indicates a centralized command-and-control structure.

An important source of change in software projects is developer turnover and this phenomenon is likely to influence the evolution of the developer network’s structural properties. Open-source software (OSS) projects are unique in that their organizational structure is predominantly self organizing, and they often lack a traditional software-engineering process that supports coordination (DiBona et al, 1999; Mockus et al, 2002; Mauerer and Jaeger, 2013). Conceptually, the lack of a centrally prescribed organizational structure enables OSS projects to more easily adapt to evolutionary pressures (Sosa et al, 2004; Kotter, 2014). One such evolutionary pressure is generated by *developer turnover*: the process where developers withdraw from a project and new developers join. In OSS projects, turnover exists in an extreme variety because the majority of developers are peripheral and characteristically volatile (Mockus et al, 2002; Crowston and Howison, 2005; Koch, 2004). In general, developer turnover is recognized as a severely detrimental process that presents a number of well-known risks to the success of a software project (Boehm, 1989). Curiously, in large OSS projects the harmonious coexistence of a large volatile peripheral group and a comparatively minuscule core group is ubiquitous. For this reason, we focus attention on specific adaptations in the coordination structure that support the organization’s ability to benefit from the human resource of peripheral developers, which are extremely abundant in comparison to core developers, while at the same time, mitigating the risks implied by peripheral developers’ volatility. Specifically, we explore whether the structural features observed in the developer network enable OSS projects to benefit from a large, but unstable, peripheral developer group. In this work, we do not posit that turnover is a strictly positive or negative phenomenon, instead we direct our investigation to the relationship between developer turnover and the structural features of developer coordination. Furthermore, we utilize the concept of core and peripheral developers to provide insightful context for the explanation of evolutionary adaptations in the developer networks’ structure.

Capturing the evolution of developer coordination is challenging and demands advanced techniques from software repository mining and dynamic systems analysis. In our approach, we make use of sophisticated information retrieval methods to determine where coordination requirements exist. Next, we apply a sliding-window technique to transform the discrete software changes recorded in the version control system into a stream of evolving developer networks. Finally, we analyze the stream of developer networks with statistical network analysis techniques to quantify the structural properties in a time resolved manner.

By means of a longitudinal empirical study on the evolution of 18 well-known OSS projects, we will address the following two main research questions (RQ):

**RQ1: Change**—*What evolutionary adaptations are observable regarding the three organizational principles in successful and long-lived open-source software projects?*

**RQ2: Growth**—*What is the relationship between the three organizational principles and project scale?*

As key results of our study, we found that the structure of developer coordination is scale free when a project contains a large number of developers and primarily during temporal periods with project growth. With respect to modularity, developer coordination becomes increasingly modular over time until an upper bound is reached. Regarding hierarchy,

developers form a hierarchical coordination structure in the early phases of a project, but the hierarchy diminishes overtime until only core developer are hierarchically arrange. Finally, we found that peripheral developers have significantly higher turnover rates compared to core developers and the adaptations seen in the developer coordination structure help to mitigate the risks implied by the abundant, but highly volatile, peripheral developer group.

In summary, we make the following contributions:

- We define a general approach for enhancing developer coordination networks with artifact-coupling information that is applicable to a wide variety of software projects.
- We adapt a previously validated network-construction approach to generate and analyze a stream of evolving developer networks by applying a sliding-window technique to extract operational data from a version control system.
- We apply our approach to study the entire publicly available history of 18 popular OSS projects that have long and complex histories, some of which date back more than 23 years.
- We extend the notion of core and peripheral developers to a network analytic perceptive and present empirical evidence that core developer groups are significantly more stable than peripheral developer groups.
- We demonstrate that developers form structures that are statistically improbable to occur in random networks, which indicates the presence of non-random organizational principles.
- We identify and discuss a number of general evolutionary trends that describe how developer coordination evolves in software projects with respect to scale freeness, modularity, and hierarchy.

Additionally, we improve on a number known methodological deficiencies of prior empirical research performed on OSS projects noted by Crowston et al. (Crowston et al, 2012). These inadequacies stem from the following: drawing conclusions from the analysis of very few (often only one) projects, a lack of attention to the early transitional phases by focusing primarily on successful projects after they have become well established, the use of coarse grained data to draw conclusions, and a general lack of longitudinal studies that explore the different phases of evolution.

All experimental data and source code are available at a supplementary website:  
<http://siemens.github.io/codeface/ese/>.

## 2 Background

We now discuss how to express developer coordination requirements as a socio-technical network, and then introduce the notion of core and peripheral developers. We conclude the section with an introduction to the concepts and definitions for scale-free networks, network modularity, and network hierarchy.

### 2.1 Developer Networks

In social sciences, networks are frequently used as a mathematically convenient structure to study the relationships between a set of actors involved in a mutual activity. In this sense, developer networks are socio-technical networks, where the mutual activities are technical in nature and stem from the software-development process. The purpose of constructing a developer network is to obtain an authentic representation of the developer-coordination

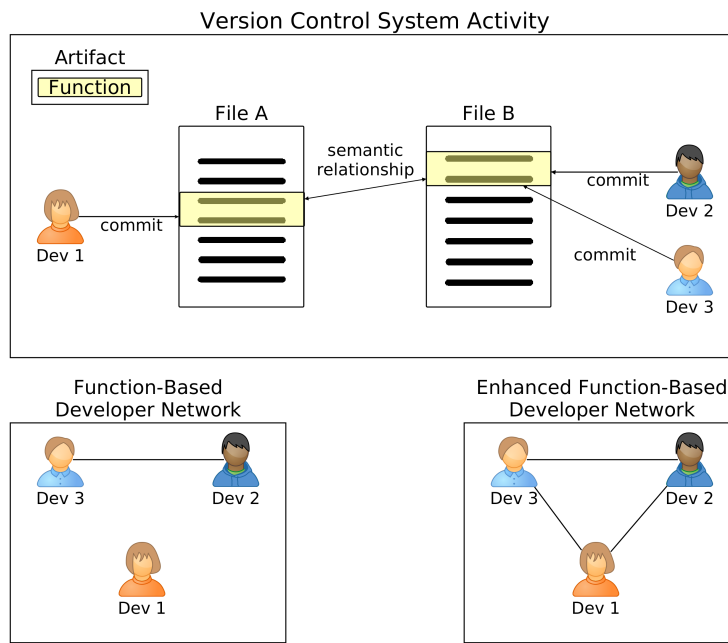


Fig. 1: Three developers edit two semantically coupled functions in separate files (top). The resulting developer network from applying the original construction method (bottom left). The resulting developer network from applying our enhanced construction approach that includes the coupling between artifacts (bottom right).

requirements implied by their development activities (Meneely and Williams, 2011; Cataldo et al, 2006; Joblin et al, 2015; Begel et al, 2010). A common approach is to construct a developer network based on the developers' contributions to software artifacts by extracting operational data from a version-control system (Joblin et al, 2015; López et al, 2006; Martinez-Romo et al, 2008; Jermakovics et al, 2011). In this type of network, the nodes represent developers and edges placed between developers represent mutual contributions to common artifacts. One drawback of these approaches is that they neglect the semantic relationships between artifacts. There are several possible heuristics, with varying degrees of precision, to determine when two developers have a coordination requirement.

### 2.1.1 Artifact Contribution

The most popular family of heuristics is based on contributions to a common artifact (e.g., files or functions) (López et al, 2006; Jermakovics et al, 2011; Martinez-Romo et al, 2008; Meneely et al, 2008; Meneely and Williams, 2011). The rationale is that an artifact is an abstraction of the software system that represents a bundle of cohesive functionality, and developers are constrained to coordinate by virtue of the interdependencies that exist inside the artifact. The chosen granularity of the artifact has implications on the authenticity of the resulting developer network. For example, a course granularity (e.g., files) results in identifying more relationships between developers, but can decrease precision by identifying relationships that are not reflective of reality. In contrast, a fine granularity (e.g., functions) will increase

precision, but may omit some developer relations. As illustrated in the bottom left of Figure 1, we use the fine-grained heuristic of contributions to common functions for identifying developer coordination. This particular heuristic has shown promise in constructing accurate developer networks and was validated by surveying open-source developers regarding the network’s correctness with respect to reflecting the developers’ perception (Joblin et al, 2015).

### 2.1.2 Artifact Coupling

Software systems are intrinsically coupled with complex associations between their artifacts (Arias et al, 2011). Through artifact coupling, developer tasks become interdependent, because changes to one artifact can propagate along the coupling relationship, and developer coordination is required to manage unintended ripple effects (Arnold and Bohner, 1993). For this reason, we make use of concepts from software-impact analysis (Arnold and Bohner, 1993) and augment developer networks with software-coupling information to get a more complete representation of the coordination requirements. The upper portion of Figure 1 illustrates the common situation where developers contribute to code that is related, but contained in separate files. As shown in the bottom left of Figure 1, without considering the relationship between the artifacts, the coordination requirement is missing between developer 1 and the remaining developers.

A diverse set of coupling mechanisms exist in software systems (e.g., function calls, class inheritance, data dependency etc.). For our study, we are most interested in coupling relationships that reflect developers’ mental model because developers will inherently rely on their internal understanding of the system to coordinate their work with others. It has been shown that traditional coupling mechanism, such as function calls, do not closely resemble developer perception (Bavota et al, 2013) and traditional coupling mechanisms are typically more obvious from the developer’s perspective because they are explicitly expressed in the source code (Cataldo et al, 2009). In our study, we make use of semantic coupling information because relationships at the semantic level are more likely to reflect artifact relationships in the developer’s mental model than other forms of coupling and for this reason is a better indicator of where coordination requirements exist.

Techniques for extracting coupling relationships that make use of information-retrieval methods have shown promise in raising the coupling concept to a semantic level that agrees with developer perception (Bavota et al, 2013). In our approach, we utilize the concept of *semantic coupling* to operationalize artifact coupling. From a purely practical perspective, semantic coupling is also well suited for our study, because it is programming-language independent, allowing the comparison between projects implemented in different programming languages. Semantic coupling is based on the principle that domain knowledge is embedded in the textual content of the software’s implementation (i.e., source code and comments) and artifacts implementing related domain concepts will share a common vocabulary (Poshyvanyk et al, 2009). At a high level, the challenge of quantifying semantic coupling depends on the identification of key terms from the overall implementation vocabulary that can be used to discriminate between distinct domain concepts. We applied *latent semantic indexing* to extract the semantic relationships between the functions that compose the software system and used cosine similarity in the latent space to determine whether two functions are semantically coupled. A detailed description of our approach is contained in the Appendix.

## 2.2 Core and Peripheral Developers

All software projects face the situation that developers withdraw at some point and need to be replaced by new, often less experienced, developers. This process of *developer turnover* is known to present enormous risks to commercial projects, because crucial knowledge is often lost with departing developers (Boehm, 1989; Mockus, 2010; Huselid, 1995). Another consequence of turnover is that replacement developers initially require mentorship, thereby consuming additional human resources by placing a burden on more experienced developers in the project. This is one factor that contributes to the well-known phenomenon that adding developers to a late project causes further delays (Brooks, 1995). In OSS projects, developer turnover exists in an extreme variation because the majority of developers have occasional, short-term participation, and generally only a very small number of core developers have consistent long-term participation (Mockus et al, 2002; Crowston and Howison, 2005; Koch, 2004). It is extraordinary that OSS projects are able to thrive under the extreme conditions of high developer turnover. For this reason, we dedicate attention to study the evolutionary pressures caused by the significantly different turnover rates between core and peripheral developers.

*Core developers* are characterized by prolonged, consistent, and intensive participation in the project, and they often have extensive knowledge of the system design and strong influence on project decisions (Mockus et al, 2002). In contrast, *peripheral developers* are characterized by irregular, and often short-lived, participation in the project. The peripheral developer group is the larger of the two, by a significant margin, but core developers are responsible for doing most of the work (Mockus et al, 2002; Crowston and Howison, 2005). While peripheral developers are an abundant human resource, they also introduce risk and consume resources. For example, empirical evidence indicates that changes made by peripheral developers introduce more architectural complexity than changes made by core developers (Terceiro et al, 2010). Regarding turnover, researchers have shown that developer turnover negatively impacts code quality, in terms of bug density (Foucault et al, 2015). Therefore, a stable and knowledgeable core developer group is imperative for ensuring system integrity in the presence of potentially inadequate changes introduced by peripheral developers. However, it appears to be ubiquitously true that successful OSS projects are capable of benefiting from a large number of volatile peripheral developers, while at the same time mitigating the associated risks. Since the coordination structure of popular OSS projects supports the symbiotic coexistence of a highly volatile peripheral developer group and a comparatively stable core developer group, we expect to observe adaptations in developer network that promote system integrity and effective coordination.

Metrics used to classify a developer as core or peripheral generally quantify the amount of participation a developer has in the project, such as lines of code or number of commits contributed (Terceiro et al, 2010; Crowston and Howison, 2005; Robles et al, 2009). A developer is then assigned to the core group if their level of participation is in the upper 20th percentile; all other developers are considered to be peripheral (Terceiro et al, 2010; Robles et al, 2009). To suit our network perspective of developer organization, we adapt the traditional participation-based definition of core developers by instead using the node degree (i.e., number of edges connected to the developer) to classify developers. Since core developers are responsible for most of the work and typically contribute to important artifacts, high degree nodes usually have high levels of participation so our operationalization of this concept is not substantially different from prior studies. Specifically, we classify a developer as core if their degree exists in the upper 20th percentile. Developers with a non-zero degree in the lower 80th percentile are peripheral developers. We consider developers with a zero

degree to be distinct from the peripheral group since these developers tend to be very inactive, often with only a single contribution.

Our study includes the notion of core and peripheral developers for two purposes. First, we know from prior empirical studies (Robles et al, 2009; Crowston and Howison, 2005; Terceiro et al, 2010) that the core developer group is typically more stable than the peripheral group. We can use this a-priori knowledge to test for evidence that the developer network structure correctly captures the notion of core and peripheral developers in the node degree. Second, we can make use of the classification of nodes to provide insightful context for the explanations of the evolutionary adaptations that we observe in the networks' structure.

### 2.3 Scale-Free Networks

Early research characterized the topology of complex networks according to the Erdős-Rényi (ER) model for random graphs, in which edges are independent and identically distributed with a fixed probability (Erdős and Rényi, 1959). In this model, the network edges are distributed according to a binomial distribution. This results in the existence of a "typical" node that is representative of most nodes in the network, and it is extremely rare to observe hub nodes with significantly more connections. More recently it has been shown that many real-world networks from fundamentally different domains (e.g., biology, sociology, scientific paper authorship, Internet routers) exhibit a substantially more organized structure than initially expected (Jeong et al, 2000; Bernard et al, 1988; Barabási et al, 2002; Dorogovtsev and Mendes, 2013). This class of networks obey a power-law degree distribution and are referred to as "scale-free". In this model, there is no notion of a typical node; hubs with many more connections than the average are common. *Scale-free* networks exhibit a structure that is the product of organizing principles that are far from uniform randomness. In the literature, it is hypothesized that scale-free networks grow according to the organizational principle of *preferential attachment*, according to which nodes entering the network have a bias to attach to already well-connected nodes (Barabási and Albert, 1999). The model of preferential attachment is only one of many possible explanations for the formation of scale-free networks (Dorogovtsev and Mendes, 2013). However, preferential attachment could explain the evolution of OSS projects because it is plausible that new developers with little experience require mentorship from highly knowledgeable core developers or core developers supervise important parts of the system, which make it necessary for peripheral developers to coordinate with them. These conditions would then lead to a situation of preferential attachment and result in a scale-free network.

We are particularly interested in the scale-freeness property of developer networks because it has a number of beneficial characteristics, including robustness to perturbations (Dorogovtsev and Mendes, 2013). This means that a random removal of a node is unlikely to disturb the connectivity of the network (e.g., fracturing the network into isolated subgraphs). In the case of a software project, robustness indicates that the withdrawal of a random developer should not severely destroy the topology of the network (i.e., the organizational structure). However, it is important to recognize that a network can only be optimized to be robust for a particular removal mechanism. As mentioned, scale-free networks are extremely robust to *random removals*, but the compromise is that they are extremely vulnerable to targeted removals. That is to say, a removal of only a small number of hub nodes can completely destroy the network topology. The question is then, does a scale-free topology offer beneficial characteristics for an OSS project? The answer to this depends on relative likelihood for withdrawal of core developers (i.e., hub nodes) and peripheral developers (i.e., low degree



nodes). If core developers are indeed less likely to leave the project compared to peripheral developers, then a scale-free topology is beneficial since the removal process does not target hub nodes. We will specifically examine these relative turnover rates in core and peripheral developers to determine whether a scale-free network increases the robustness of the collaborative structure.

To identify a scale-free network, one must show that the degree distribution is described by  $p(k) \propto k^{-\alpha}$ , where  $p(k)$  is the probability of observing a node with  $k$  connections and  $\alpha$  as the power-law scaling parameter. In Section 3.4, we will discuss in detail how to determine the scaling parameter  $\alpha$  and verify whether the model accurately describes the observed network.

## 2.4 Network Modularity

The scale-freeness property is a statement about individual nodes and their respective degrees, but it entirely neglects features regarding the connectivity of a node's local neighborhood. In our case, the local neighborhood of developer  $X$  is a subnetwork that represents all coordination requirements between all developers that are connected to developer  $X$ . *Modularity* captures the connectivity of a node's local neighborhood by quantifying the extent to which nodes form connected groups. In the analysis of software artifact relationships, coupling (the extent to which an artifact is externally dependent) and cohesion (the extent to which an artifact is internally dependent) are frequently used. Modularity is expressed as a relationship between the concepts of coupling and cohesion such that a highly modular structure is one that exhibits low coupling and high cohesion. In a social network, modularity is high when the neighbors of node  $i$  have edges to other neighbors of node  $i$ , which is called a *triadic closure*. The conjecture is that, for three nodes  $X$ ,  $Y$  and  $Z$ , the edges  $X - Y$  and  $X - Z$  will imply edge  $Y - Z$  to exist by virtue of the commonality from both  $Y$  and  $Z$  being connected to  $X$ . This natural clustering has been shown to exist in many real-world networks (e.g., it indicates specialization of function in biological networks or people with common interests in social networks) (Dorogovtsev and Mendes, 2013). Similarly, in developer networks, we expect modularity to arise from specialization in the developer roles and contributions to interdependent tasks. This expectation follows from Conway's law, which hypothesizes that the modular structure of a software system should be reflected in the developer organization (Conway, 1968).

To quantify modularity, we use the well studied *clustering coefficient*:

$$c_i = 2n_i / k_i(k_i - 1), \quad (1)$$

where  $n_i$  is the number of edges between the  $k_i$  neighbors of node  $i$  (Boccaletti et al, 2006). The intuition is that  $k_i(k_i - 1)/2$  edges can exist between  $k_i$  nodes, and the clustering coefficient is a ratio that reflects the fraction of existing edges between neighbors divided by the total number of possible edges.

## 2.5 Network Hierarchy

So far, we have introduced scale-freeness, which describes the distribution of edges among nodes, and network modularity, which describes the grouping of nodes according to the local network structure. The concept of *hierarchy* brings these two concepts together by addressing how local groups are arranged relative to each other. In a hierarchical network, there exists

stratification within the network that stems from cohesive groups being embedded within larger and less cohesive groups. This stratification is manifested as a relationship between the node clustering coefficient and the number of connections, that is, the node degree (Ravasz and Barabási, 2003). Nodes with high degree and low clustering coefficient represent the top of the hierarchy; nodes with low degree and high clustering coefficient are located at the bottom of the hierarchy. The relationship between node degree and clustering coefficient in a hierarchical network is described by  $c(k) \propto k^{-1}$ , where  $c(k)$  is the clustering coefficient for a node with degree  $k$  (Ravasz and Barabási, 2003). To test for the presence of hierarchy, we applied a robust linear regression technique to solve for the optimal linear model satisfying the functional form  $Y = \beta_0 + \beta_1 X$ , where the clustering coefficient is the response variable denoted by  $Y$  and node degree is the predictor variable denoted by  $X$ . If the optimal linear model has a nonzero slope (i.e.,  $\beta_1 < 0$ ) and the slope parameter is statistically difference from zero, such that  $p < 0.05$  where  $p$  is that probability that  $\beta_1 = 0$ , we can conclude that hierarchy is present.

Intuitively, this relationship implies that nodes of high degree tend to be connected to many different groups that are themselves loosely coupled to each other. What makes hierarchy particularly interesting is that it is not explained solely by preferential attachment and therefore indicates an entirely separate organizational principle (Ravasz and Barabási, 2003). Hierarchy in developer networks indicates the existence of an organizational structure that transcends the local network structure and represents an organizational element that sprawls the network topology at different layers of abstraction to improve coordination between developers that are members of different groups (Hinds and McGrath, 2006).

### 3 Methodology

We now discuss our experimental methodology to study the evolutionary principles of developer coordination in OSS projects. The study is divided into two parts. First, we construct a series of developer networks using historical data stored in the version-control systems of a set of sample projects. Second, we apply network-analysis techniques to examine the network topology (scale-freeness, modularity, and hierarchy) as a function of time and relate the measurements to developer turnover rates in core and peripheral developer groups.

#### 3.1 Developer Network Construction

In software projects, developers often work on distinct functions that are conceptually related to other functions that are written and maintained by other developers. For example, one developer provides a set of functions to perform logging of user information and a second developer makes extensive use of these functions for logging user triggered events in the user interface. The consequence of constructing developer networks based only on mutual contributions to a single artifact is that higher order developer coordination requirements, which stem from contributions to artifacts that are conceptually related but logically separated, will be omitted. To capture a more complete model of the developer coordination, we improve on the traditional approaches (López et al, 2006; Jermakovics et al, 2011; Martinez-Romo et al, 2008; Meneely et al, 2008; Meneely and Williams, 2011; Joblin et al, 2015) by augmenting developer networks with artifact-coupling information. Our approach is inspired by the framework proposed by Caltaldo et al. (Caltaldo et al, 2008) for identifying coordination requirements between developers working on interrelated tasks. While our implementation

is one variation that uses a function-level artifact and semantic coupling, this approach naturally extends to accommodate other artifacts (e.g., configuration files, requirements, documentation etc.) and coupling mechanisms (e.g., dynamic, structural, co-change etc.). Specifically, we reconcile the developer-artifact contribution (see Section 2.1.1) and software-artifact coupling (see Section 2.1.2) information as follows: We begin by first identifying all developers' contributions to all functions in the system and express the contributions of  $M$  developers to  $N$  functions in an  $M \times N$  matrix as

$$A_{\text{contrib}} = \begin{bmatrix} f(d_1, a_1) & \dots & f(d_1, a_N) \\ \vdots & \ddots & \vdots \\ f(d_M, a_1) & \dots & f(d_M, a_N) \end{bmatrix}, \quad (2)$$

where  $A_{\text{contrib}}$  is the function-contribution matrix and  $f(d_i, a_j)$  represents the magnitude of contribution to artifact  $a_j$  by developer  $d_i$ . In our study, we quantify the magnitude of contribution to an artifact based on the number of commits made by the developer to a specific function.

Next, we compute a matrix that represents the semantic coupling between artifacts using the approach described in Section 2.1.2. We represent the coupling for  $N$  artifacts in an  $N \times N$  matrix as

$$A_{\text{coupling}} = \begin{bmatrix} \phi(a_1, a_1) & \dots & \phi(a_1, a_N) \\ \vdots & \ddots & \vdots \\ \phi(a_N, a_1) & \dots & \phi(a_N, a_N) \end{bmatrix}, \quad (3)$$

where  $A_{\text{coupling}}$  is the artifact-coupling matrix and  $\phi(a_i, a_j) = 1$  if the two artifacts  $a_i$  and  $a_j$  are coupled and  $\phi(a_i, a_j) = 0$  otherwise. In our study, this matrix represents the semantic coupling between functions.

Finally, we combine the information contained in both matrices using the following matrix multiplication operation

$$D_{\text{coord}} = A_{\text{contrib}} \times A_{\text{coupling}} \times A_{\text{contrib}}^{\top}, \quad (4)$$

where  $D_{\text{coord}}$  is the developer-coordination matrix, with elements that represent the magnitude of the coordination requirement between two developers. Intuitively, the operation expressed in Equation 4 is computing the developers' mutual dependence as the sum of contributions to common artifacts and artifacts that are semantically coupled. The resulting matrix  $D_{\text{coord}}$  is symmetric with respect to the principal diagonal and represents the adjacency matrix for a weighted and undirected developer network.

### 3.2 Developer Network Stream

In a second step, we capture the time-resolved evolution of developer organization by applying a graph data-stream model to the network-construction procedure; a project's history is segmented into sequential overlapping observation windows, where each observation window captures a finite range of development activity. To linearize the development history, we flattened the master branch of the Git version control system, which is essentially the linearization of a directed acyclic graph. All commits were then temporally ordered using the commit time. The  $n^{\text{th}}$  observation window is defined as a set  $W_n$  of commits, such that  $W_n = \{\text{commit}_t \mid t \in [t_0 + n \cdot \Delta_{\text{step}}, t_0 + n \cdot t_{\text{step}} + \Delta_{\text{window}}]\}$ . Where  $\text{commit}_t$  is the commit

occurring at time  $t$ ,  $t_0$  is the time of the initial commit,  $\Delta_{\text{window}}$  is the window size, and  $\Delta_{\text{step}}$  is the step size. Since software projects typically have long term trends (e.g., number of contributing developers), the evolution is temporally dependent and must be treated as a nonstationary process. This implies that the statistics (e.g., mean and variance of the metrics) will vary depending on when the project is observed. To properly analyze project evolution, we use a small enough observation window (90 days) for which the development activity has been shown to be quasi stationary (Meneely and Williams, 2011)—a technique that is frequently employed in other domains with temporally-dependent processes (Huang et al, 1998). To avoid artifacts that arise from aliasing and discontinuities between the edges of the observation windows, we opted for an overlapping-window technique (Huang et al, 1998) with a step size that is half of the window size. While smaller step sizes may be better, because of greater temporal resolution, we observed that using a smaller step size did not change the results, but did significantly add to the computational costs. In contrast, increasing the step size so that the windows did not overlap obscured periodic components in the data.

For each time window, we construct a network to represent the topology of developer coordination during a finite time range. The sequence of all finite windows generates the graph stream capturing the dynamic evolution of developer coordination over the entire project history. Each graph stream is then processed to extract a multivariate time series composed of the measurements that quantify the concepts of scale-freeness, modularity, hierarchy, in addition to other context features, such as network size.

### 3.3 Developer Transitions

Developer turnover — the process by which developers enter and withdraw from a project — provides important insights into the stability of the organizational structure of a project. Instead of looking only at developer turnover, we expand on this concept by looking at how developers transition from different levels of participation. Particularly, we employ sequential-data modeling techniques to formally address this aspect of the network evolution. We make use of the discrete state Markov model (Bishop, 2006) by assigning a discrete state to every developer in the project for each time window. A developer is able to occupy one of the following four possible states.

*Core*: developers with degree in the upper 20th percentile

*Peripheral*: non-core developers with non-zero degree

*Isolated*: developers with a zero degree

*Absent*: developers that did not make any commits

Each developer’s state transitions are expressed by a sequence of random variables  $X_t \in \{s_1, s_2, s_3, s_4\}$  that can take on any of the four states. We then employ the Markov property such that  $\Pr(X_{t+1} = x | X_1 = x_1, X_2 = x_2, \dots, X_t = x_t) = \Pr(X_{t+1} = x | X_t = x_t)$ . The assumption is that, to determine the next state transition, only information about the previous state is required. Using this assumption, we are able to represent developer transitions from state to state as an  $N \times N$  transition matrix, in which each element indicates the probability of transitioning from any state in the state space  $N$  to any other state during the entire project’s evolution. We used maximum-likelihood estimation to solve for each state transition parameter (Bishop, 2006). Figure 2 provides an example developer transition Markov chain: The core developers stay in the core state in the following release with a 54% probability, transition to the peripheral state with 25% probability, with 4% probability transition to the isolated state, and with 17% probability to the absent state. All transition probabilities are

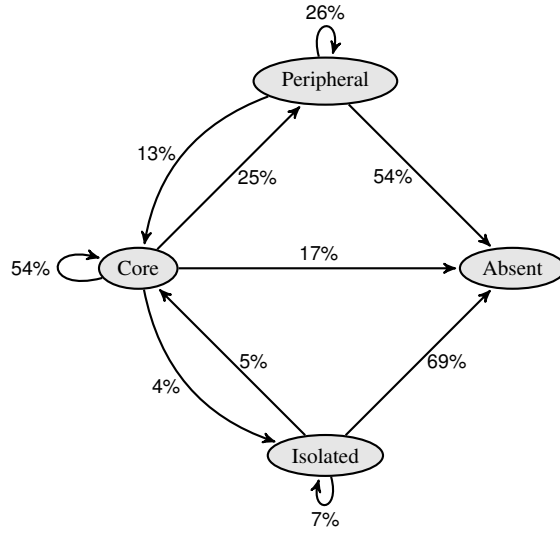


Fig. 2: The developer-group stability for QEMU shown in the form of a Markov Chain. A few less important edges have been omitted for visual clarity.

between 0 and 1, and the sum of all transitions from a single state is equal to 1, to ensure that the conditions for a probability function are maintained.

### 3.4 Complex Network Topology

To determine whether a network is scale free, one needs to show that the degree distribution is explained by a power law (Albert and Barabási, 2002). There are a number of frequently experienced pitfalls in trying to test a network for scale freeness, and for this reason, we dedicate significant effort to ensuring statistical rigor. A necessary but not sufficient condition for power-law behavior is that the log-log scaled degree distribution is described by a linear relationship, which makes identifying a power law involved and challenging (Clauset et al, 2009; Goldstein et al, 2004). Further complicating the situation, it is often the case that empirical data exhibit a power law only in the tail of the degree distribution, so the power-law model is rarely valid for the entire set of observations.

In our study, we use a maximum-likelihood technique to solve directly for the power-law model parameters. Then, we perform a Kolmogorov-Smirnov goodness-of-fit test on the fitted model. For the moment, let us assume that the lower bound for the power law,  $k_{\min}$ , is known, then the power-law scaling parameter,  $\alpha$ , can be solved using the following approximation to the maximum-likelihood estimate:

$$\hat{\alpha} \simeq 1 + n \left[ \sum_{i=1}^n \ln \frac{k_i}{k_{\min} - \frac{1}{2}} \right]^{-1}, \quad (5)$$

where  $n$  is the number of nodes in the network and  $k_i$  is the degree of node  $i$  (Clauset et al, 2009). The choice of  $k_{\min}$  is critical because choosing too low a value will bias the result by trying to fit a power law to data that do not obey a power law. In contrast, too large a value

results in throwing away useful samples for estimating  $\alpha$  and will increase the statistical error. We solve for the optimal  $k_{\min}$  iteratively by selecting a value, solving for  $\alpha$  using Equation 5, and then testing the fit using the Kolmogorov-Smirnov (KS) statistic. The optimal value for  $k_{\min}$  is then chosen based on the best fit according to the KS statistic.

Once we have solved Equation 5 for the best-fit power law, we still do not know whether a power law is a plausible model for describing the observed data. For this purpose, we perform a goodness-of-fit test. This test will discern whether the discrepancy can be explained by finite random sampling or because the power law is not an appropriate model. To perform the test, we generate an ensemble of synthetic data sets by sampling the fitted power law.<sup>1</sup> Then, we fit a power law to the synthetic data sets. Next, we compute the KS statistic between all data sets and their corresponding fitted power laws. Finally, we compute a  $p$  value that represents the fraction of KS statistics from the synthetic data sets that exceed the empirical one. A large  $p$  value indicates that the deviation of the observed data from the fitted model can be attributed to statistical fluctuation and not to a systematic error from the selection of an inappropriate model. For  $p < 0.05$ , we reject the null hypothesis that the observed data are described by a power law.

When there are too few samples for statistical tests to be reliable, which is common early in the project history, we instead use the Gini coefficient (Atkinson, 1970) to characterize the amount of inequality in the network's degree distribution. The Gini coefficient is bounded between 0 and 1, where 1 indicates strong inequality (i.e., possibly scale free); 0 indicates strong equality (i.e., not scale free). By definition, scale-free networks contain hub nodes, and as a result there is strong inequality in the distribution of edges connecting nodes. From this we conclude that a high Gini coefficient is a necessary condition for scale freeness, and if the network has a low Gini coefficient, it cannot be scale free.

## 4 Study & Results

### 4.1 Hypotheses

We now present and discuss our four hypotheses regarding the evolution of developer coordination networks.

**H1**—*Core developers (those with a node degree in the upper 20th percentile) exhibit significantly more stability than peripheral developers (those with a node degree in the lower 80th percentile).*

When developers withdraw from a project, there are potentially severe consequences as a result of the loss of knowledge and the additional resources required to mentor replacement developers. However, many successful OSS projects have adapted to benefit from an abundant supply of a group of peripheral developers that is inherently unstable in comparison to the group of core developers. We expect that the distinct characteristics of the core and peripheral developer groups are responsible for some of the observable structural features in the developer networks.

**H2**—*For the growth of a large project to be sustainable over a long period of time, the developer coordination structure must become scale free.*

<sup>1</sup> We chose the number of synthetic data sets to generate to introduce a precision tolerance of two decimal places in the  $p$  value.

Coordination of a large number of developers demands specialized coordination mechanisms, because the number of potential interactions among developers is quadratic in the number of developers (Brooks, 1995). Additionally, since the peripheral developer group, representing the majority of OSS developers, is conjectured to be unstable, the implication is that a healthy developer network must be robust to node removals. Therefore, we expect that large developer groups self-organize into scale-free networks as an optimization for mitigating the coordination overhead and achieving resilience to coordination breakdowns (see Section 3.4). Following the reasoning of Brooks (Brooks, 1995), as the developer network grows, we expect, at some point, the developer count should stagnate or decrease, because of ineffective coordination leading to a loss of productivity and developers' motivation to participate. It has been shown that, in some situations, Brooks' law does not apply to OSS projects (Koch, 2004), and we hypothesize that scale freeness is a reasonable principle to explain this observation. Therefore, we expect very large projects to exhibit the scale-freeness property as a mechanism to maintaining productivity despite the potentially enormous coordination costs and risks imposed by a large but unstable peripheral developer group. Finally, scale freeness is an emergent property of a self-organizing system that is motivated by necessity. Since small developer groups do not benefit from a scale-free network structure as much as large developer groups, we do not expect small projects with a small number of developers to form scale-free networks.

**H3**—*Developers initially form loosely coordinated groups that are not internally well connected (i.e., low modularity). As time proceeds, developer groups tend to become more strongly coordinated in terms of the clustering coefficient until an upper bound is reached.*

As a project evolves, several factors encourage developers to coordinate, but there are also opposing forces. For example, based on prior experience and empirical evidence, software evolution tends to cause an increase along several project dimensions (e.g., lines of code, complexity, number of developers etc.) and will demand increasing levels of coordination between developers to avoid system degradation (Lehman et al, 1997; Lehman and Ramil, 2001). Furthermore, it is reasonable to expect that developers will become more familiar with each other and rely on the external knowledge of others for support in the completion of development tasks. Empirical evidence from studies on various OSS projects also suggests that developers tend to specialize on particular artifacts (e.g., subsystems or files) and form groups with common responsibilities and shared mental models (Koch, 2004; Joblin et al, 2015). These influences increase modularity in the developer network by causing additional edges to form in local sub-networks that are dedicated to a particular responsibility. The opposing force arises from the quadratic scaling between the number of developers and potential coordination relationships, where the cost of coordination can easily dominate the benefit achieved from coordination (Brooks, 1995). Therefore, developer coordination is constrained to evolve in a manner that balances these opposing forces. We expect that an equilibrium exists between the benefit and cost of coordination, and this will govern the evolution of developer coordination.

**H4**—*In early project phases, the developer-coordination structure is hierarchically arranged. As a project grows and matures, the developer-coordination structure will gradually converge to a network that does not exhibit hierarchy, as the command-and-control structure becomes more distributed.*

In the early phases of a project, it is conceivable that the small number of initial developers have a comprehensive understanding of the global project details and are capable of coordinating the work with others in a centralized configuration. In these early project conditions,

hierarchy is an effective organizational structure because it promotes efficiency through regularity and is appropriate when the developer network is stable (Kotter, 2014). As the project evolves and grows in the number of developers and system size, developer coordination becomes increasingly formidable, especially, once the peripheral developer group has grown to be significantly larger than the core developer group. Empirical evidence indicates that efficiency in large OSS projects is the result of self-organizing cooperative and highly decentralized work (Koch, 2004), which becomes increasingly important as a project grows. The result is that the command-and-control structure must evolve to become more distributed, because no single person could reasonably have a comprehensive understanding of the global project state, and distributed self-organization must take over. Furthermore, hierarchy is an intrinsically inflexible organizational structure that strongly promotes regularity (Kotter, 2014), but as the project evolves, organizational flexibility becomes increasingly important so that the project can avoid the detrimental misalignment of organizational structure and the technical structure as a result of evolution (Sosa et al, 2004).

## 4.2 Subject Projects

For the purpose of our study, we selected 18 OSS projects as listed in Table 1. The projects vary in the following dimensions: (a) size (source lines of code, from 50 KLOC to over 16 MLOC, number of developers from 25 to 1000), (b) age (days since first commit), (c) technology (programming language, libraries), (d) application domain (operating system, development, productivity, etc.), and (e) version-control system used (Git, Subversion). We chose these projects because they are all widely deployed, and have long development histories. The data and list of figures for all projects are available at the supplementary website.

Table 1: Overview of subject projects

Project	Domain	Lang	SLOC	Commits	Devs	Period
Apache HTTP	Server	C	2M	73K	26	05/99–06/15
Chromium	User	C/++, JS	16M	533K	1056	07/08–06/15
Django	Devel	Python	400K	38K	105	07/05–01/15
Firefox	User	C/++, JS	12M	230K	474	03/98–06/15
GCC	Devel	C/++	7M	137K	122	06/91–01/15
Homebrew	User	Ruby	100K	42K	525	05/09–06/15
Joomla	CMS	PHP	400K	20K	78	09/05–06/15
jQuery	Devel	JS	65K	12K	30	03/06–06/15
Linux	OS	C	17M	570K	1512	04/05–05/15
LLVM	Devel	C/++	1.2M	120K	128	06/01–06/15
Mongo	Database	C/++, JS	600K	28K	53	10/07–06/15
Node.js	Devel	C/++, JS	5M	23K	53	04/09–05/15
PHP	Devel	PHP, C	2.5M	100K	66	04/99–05/15
QEMU	OS	C	1M	37K	157	11/05–06/15
Qt 4	Devel	C++	1.5M	36K	122	03/09–04/15
Rails	Devel	Ruby	200K	49K	213	11/04–06/15
Salt	Devel	Python	200K	44K	205	02/11–06/15
U-Boot	Devel	C	1.2M	32K	134	12/02–06/15



### 4.3 Developer-Group Stability

To address H1, we now present the results regarding the stability of core and peripheral developer groups. We applied the procedure described in Section 3.3 to construct a Markov chain representing the transitions between the four possible developer states (core, peripheral, isolated, and absent). Considering the evolution of all subject projects, the primary finding is that core developers are significantly less likely to withdraw compared to non-core developers. In Table 2, the Markov chain for each project is presented. The Markov chain for QEMU will be described as a representative of the primary result. Reading across the row labeled “core” of the table for QEMU, we see that core developers are most likely to stay in the core state from one development cycle to the next with 83.7% probability. The next most likely state is the peripheral state with 15.8% probability, and finally there is the low probability of 0.5% that a core developer becomes absent. In comparison to developers in the peripheral state and isolated state, the probability of becoming absent is 10.9% and 16.4% respectively. This result is convincing evidence that the groups of peripheral and isolated developers, or more generally non-core developers, are significantly less stable than core developers. Furthermore, we see that, once a developer enters the absent state, there is an overwhelming probability that the developer will not return to the project. This result suggests, once a developer becomes absent for a single revision, she will likely never participate in contributing code in the future. Since entering the absent state most likely indicates a total loss of the individual and any valuable knowledge they possess, the peripheral developers introduce risk through their volatility.

For all projects, the data indicates that the core developer group is significantly more stable than the peripheral developer group, and on this basis we *accept H1*.

Table 2: Probabilities of Developer State Transitions

	absent	core	isolated	peripheral
absent	0.964	0.003	0.007	0.027
core	0.017	0.617	0.010	0.356
isolated	0.322	0.022	0.291	0.365
peripheral	0.181	0.103	0.077	0.639
(a) Apache HTTP				
	absent	core	isolated	peripheral
absent	0.982	0.002	0.002	0.015
core	0.095	0.680	0.006	0.218
isolated	0.433	0.022	0.239	0.306
peripheral	0.332	0.081	0.027	0.559
(c) Django				
	absent	core	isolated	peripheral
absent	0.992	0.000	0.001	0.007
core	0.000	0.823	0.000	0.177
isolated	0.155	0.001	0.556	0.288
peripheral	0.057	0.050	0.017	0.876
(e) GCC				
	absent	core	isolated	peripheral
absent	0.979	0.002	0.004	0.016
core	0.107	0.638	0.005	0.251
isolated	0.416	0.032	0.348	0.204
peripheral	0.298	0.095	0.051	0.556
(g) Joomla				
	absent	core	isolated	peripheral
absent	0.972	0.002	0.001	0.025
core	0.053	0.681	0.001	0.264
isolated	0.401	0.016	0.184	0.399
peripheral	0.291	0.086	0.009	0.614
(i) Linux				
	absent	core	isolated	peripheral
absent	0.974	0.000	0.004	0.022
core	0.000	0.691	0.000	0.309
isolated	0.412	0.000	0.329	0.259
peripheral	0.218	0.102	0.021	0.660
(k) Mongo				
	absent	core	isolated	peripheral
absent	0.978	0.001	0.000	0.021
core	0.017	0.698	0.000	0.285
isolated	0.347	0.006	0.136	0.511
peripheral	0.170	0.089	0.003	0.738
(b) Chromium				
	absent	core	isolated	peripheral
absent	0.987	0.000	0.001	0.012
core	0.012	0.700	0.001	0.287
isolated	0.349	0.007	0.208	0.436
peripheral	0.193	0.092	0.017	0.698
(d) Firefox				
	absent	core	isolated	peripheral
absent	0.969	0.006	0.000	0.026
core	0.246	0.500	0.000	0.253
isolated	0.500	0.000	0.500	0.000
peripheral	0.433	0.095	0.000	0.473
(f) Homebrew				
	absent	core	isolated	peripheral
absent	0.980	0.001	0.001	0.018
core	0.051	0.624	0.006	0.318
isolated	0.250	0.125	0.000	0.625
peripheral	0.344	0.088	0.004	0.564
(h) jQuery				
	absent	core	isolated	peripheral
absent	0.980	0.000	0.003	0.017
core	0.014	0.719	0.002	0.265
isolated	0.351	0.015	0.224	0.409
peripheral	0.187	0.093	0.029	0.690
(j) LLVM				
	absent	core	isolated	peripheral
absent	0.972	0.002	0.003	0.023
core	0.119	0.686	0.008	0.187
isolated	0.538	0.015	0.288	0.159
peripheral	0.406	0.081	0.046	0.467
(l) Node.js				

	absent	core	isolated	peripheral
absent	0.982	0.001	0.004	0.014
core	0.021	0.672	0.008	0.299
isolated	0.353	0.022	0.323	0.302
peripheral	0.198	0.099	0.059	0.643

(m) PHP

	absent	core	isolated	peripheral
absent	0.964	0.003	0.010	0.022
core	0.103	0.564	0.030	0.303
isolated	0.395	0.048	0.349	0.208
peripheral	0.257	0.101	0.055	0.588

(o) Qt 4

	absent	core	isolated	peripheral
absent	0.986	0.001	0.001	0.013
core	0.020	0.834	0.001	0.146
isolated	0.162	0.003	0.545	0.290
peripheral	0.137	0.047	0.014	0.803

(q) Salt

	absent	core	isolated	peripheral
absent	0.991	0.000	0.001	0.008
core	0.005	0.837	0.001	0.158
isolated	0.164	0.000	0.507	0.330
peripheral	0.109	0.047	0.017	0.828

(n) QEMU

	absent	core	isolated	peripheral
absent	0.994	0.000	0.000	0.005
core	0.033	0.813	0.000	0.153
isolated	0.185	0.005	0.439	0.372
peripheral	0.147	0.045	0.010	0.798

(p) Rails

	absent	core	isolated	peripheral
absent	0.980	0.003	0.001	0.016
core	0.121	0.589	0.006	0.283
isolated	0.440	0.027	0.167	0.365
peripheral	0.341	0.096	0.030	0.533

(r) U-Boot

#### 4.4 Scale Freeness

We now discuss the results of applying the procedure described in Section 3.4 to address H2. The primary goal is to determine whether a power-law degree distribution is a plausible model to describe the observed developer networks and thus be characterized as scale-free networks. We must eliminate Apache HTTP from this evaluation because the project has too few developers, and the statistical error with small sample sizes can lead to inaccurate conclusions. For the remaining 17 projects, if the goodness-of-fit  $p$  value is greater than 0.05 and the  $k_{\min}$  parameter does not exclude more than 30 developers from the power-law model, we can confidently conclude that the network is scale free.

One primary finding, which is true for 17 subject projects, is that the scale-freeness property is temporally dependent, which means that this property is not universally present with respect to time. This is notable because it is a distinctly different view point from prior studies that approached the topic of scale freeness from a temporally static perspective (López et al, 2006). To illustrate this result, a typical chronological profile is shown in the left portion of Figure 3, taken from LLVM. The top figure illustrates the network growth in terms of the number of developers contributing to the project, and the bottom figure illustrates the Gini coefficient. Each sample point represents a measurement for a single developer network that is computed for a single development time window. The shape of the sample point represents whether the network is in a scale-freeness state during the given development window. To help draw attention to the general trends in the data, a smooth curve has been fitted using locally weighted scatterplot smoothing, with 99% confidence intervals in grey. The evolutionary profile of a project is typically composed of the following three distinct temporal phases.

The initial phase, which can last for a number of years, is characterized by extremely limited growth in the number of contributing developers. During this period, the network exhibits high levels of coordination equality, because most developers are similar with respect to the degree of coordination with other developers and so the coordination requirements

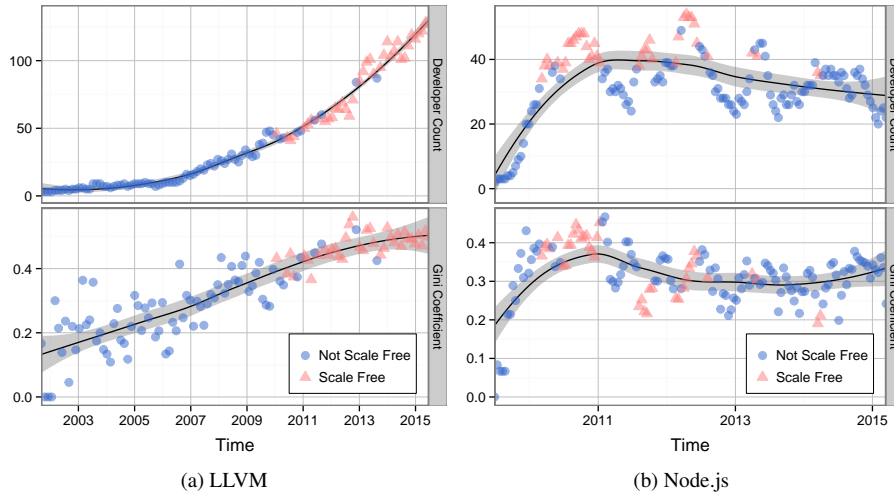


Fig. 3: Evolutionary profile for entire history of LLVM and Node.js. Time series are shown for the Gini coefficient (bottom) and the number of developers (top). A smooth curve is fitted to the observations with the 99% confidence interval shown in gray. The shape of the data points indicates whether the network was scale free for a given point in time.

are uniformly distributed among all developers. The magnitude of coordination equality in the network is quantified using the Gini coefficient of the corresponding degree distribution. This is shown in Figure 3 for LLVM, where we see an initially low Gini coefficient (i.e., high equality). A low Gini coefficient indicates that, during this initial phase, most developers are similar in their degree of coordination with others, and the network is not scale free, because it lacks the characteristic hub nodes discussed in Section 3.4.

In the second phase, we see that projects reach a critical mass point, at which a positive trend component is visible in the number of contributing developers. For LLVM (see Figure 3), this point occurred in late 2006 to early 2007. Following this point, super linear growth with an increasing slope occurs until the end of the analysis period. During that time, the Gini coefficient also has a positive trend, indicating that the network has progressively less equality, because hub nodes, with significantly more coordination requirements than the average developer, begin to form. During this phase, the scale-freeness property emerges for the first time, but the state of being scale free is initially unstable. Most of the projects become scale free once the network size has reached roughly 50 developers, but in no case does a network exceed a size of 86 developers without first becoming scale free. The only exception is for Linux because we did not have sufficient data to observe the early phases. The developer network size at the earliest point in time that the network becomes scale is shown in Table 2 for all projects under the column label “SF Size”. Overall, the two important results from this phase are that, during time periods with much less than 50 developers, developer equality is relatively high and scale-freeness is not a common property. In contrast, during periods that significantly exceed 50 developers, the developer networks are predominantly scale free. Essentially, the scale-freeness property appears to be dependent on the network size and the time of observation.

In 12 projects (Chromium, Django, Firefox, GCC, Homebrew, Joomla, Linux, LLVM, QEMU, Rails, Salt, and U-Boot), a third phase is visible in which the scale-freeness property

stabilizes and is rarely lost. In all of the projects that achieve stable scale-freeness, they demonstrate the capability of long term sustained, and often accelerating, growth in the number of contributing developers. In the other 6 projects (Apache HTTP, jQuery, Mongo, Node.js, Qt 4, and PHP), scale-freeness is either never achieved, remains unstable indefinitely, or is lost indefinitely. The growth profile in these 6 projects is very different from the projects that achieve stable scale freeness and growth appears to be unsustainable because project growth decreases with time and often the number of contributing developers even drops. An example of these two distinct cases is presented in Figure 3, where LLVM reaches stable scale freeness and has long term accelerating growth, in contrast Node.js does not achieve scale free stability and has unsustainable growth with long term loss of developers. The percentage of time each project spends in a scale-free state is shown in Table 2 under column “% SF”. The measurements indicate that the large projects that have had long term growth spend a significantly larger percentage of time in a scale-free state in comparison to projects without long term sustained growth.

A scale-free network is characterized based on whether the *tail* of the degree distribution is described by a power law (see Section 3.4). A rarely addressed yet important factor is the proportion of nodes that are described by the power law. We illustrate this point in Figure 4, where the tail region described by the power law excludes the majority of developers of the Linux kernel. We found that, in all projects, the proportion of developers that are described by the power law is low and typically ranges between 20%–50%. We saw that, there is an inverse relationship between the network size and the percentage of developers that are characterized by a power law. We illustrate the results for all projects in Table 2, where column “ $k_{min}$ ” represents the lower bound for the power-law distribution, column “% Dev SF” represents the percentage of developers the are described by the power-law distribution, and column “SF Dev” represents the absolute number of developers described by the power-law distribution. These measurements are taken from the most recent analysis period and the corresponding analysis windows are provided in Table 2.

We found two interesting outlier projects with respect to the presence of the scale-freeness property. For PHP and jQuery, the network size reached a peak at roughly 50 developers for several months yet never reached a stable scale-free state. After the peak, both projects then experienced years of continuous loss of developers and never recovered. Another interesting outlier is Firefox, where during the period of September 2009 to December 2009 the network was frequently not scale free. While the cause of the disruption is unknown, we learned that, during this period, Firefox experienced severe release problems resulting in a major revision being released one year late.<sup>2</sup> It is interesting to note that the network-structure disturbances were observable several months before the first public announcement of release problems.

In summary, our study suggests that OSS projects lack a scale-free state in the initial phases while the network size is small. In networks where growth significantly exceeds 50 developers, we always see the emergence of a scale-free developer network and no project ever grew beyond 86 developers without first becoming scale free. The caveat is that the scale-freeness property is temporally dependent, and in some projects, remains in an oscillatory state indefinitely. Overall, we *accept H2*.

---

<sup>2</sup> <http://www.cnet.com/news/mozilla-pushes-back-firefox-3-6-4-0-deadlines>

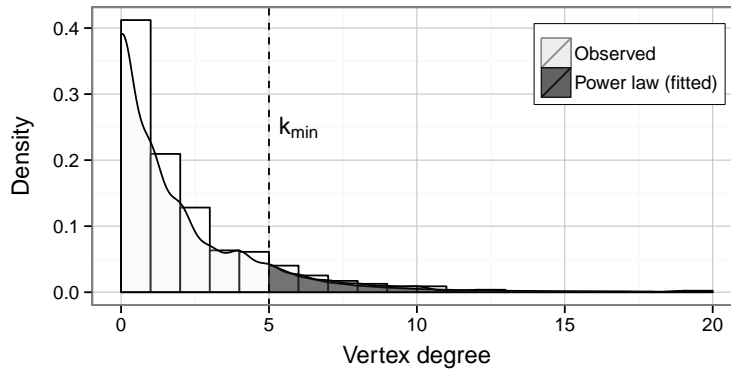


Fig. 4: Power-law fitted to degree distribution for Linux. The power-law distribution describes the developers with a degree greater than  $x_{min}$ , the majority of low degree developers are not described by a power-law.

Table 2: Developer network structural measurements

Project	Scale Free					Modularity		Hierarchy	
	S.F. size	%S.F.	$k_{min}$	% Dev S.F.	# Dev S.F.	$\mu_{cc}$	$\sigma_{cc}^2$	$\beta_{1_{early}}$	$\beta_{1_{late}}$
Apache HTTP	N/A	0.00	7	0.46	6	0.49	0.12	-1.14	-0.85
Chromium	71	97.60	1012	0.08	53	0.44	0.04	-0.40	-0.35
Django	42	25.00	49	0.60	59	0.57	0.04	-2.32	-0.51
Firefox	73	90.60	481	0.07	30	0.48	0.05	-0.43	-0.27
GCC	36	55.90	164	0.26	30	0.43	0.04	-0.73	-0.40
Homebrew	80	87.50	355	0.49	230	0.61	0.02	-1.28	-0.29
Joomla	55	20.00	2	0.66	35	0.38	0.16	-1.67	-0.66
jQuery	30	1.41	5	0.80	4	0.47	0.10	-2.09	-1.66
Linux	515	98.80	1521	0.05	69	0.58	0.05	-0.28	-0.25
LLVM	46	32.70	51	0.24	30	0.45	0.08	-2.22	-0.39
Mongo	51	3.28	30	0.67	30	0.38	0.07	-2.30	-0.53
Node.js	35	16.30	1	0.47	9	0.16	0.10	-2.30	-1.33
PHP	46	14.60	29	0.65	30	0.45	0.08	-0.85	-0.44
QEMU	37	61.00	88	0.27	31	0.53	0.05	-2.19	-0.35
Qt 4	86	43.80	3	0.86	6	0.68	0.12	-0.50	-0.92
Rails	38	69.20	55	0.21	30	0.58	0.08	-2.17	-0.34
Salt	32	82.20	89	0.36	74	0.55	0.07	-0.95	-0.31
U-Boot	41	64.60	41	0.58	66	0.60	0.05	-1.39	-0.29

S.F. size - network size at first appearance of scale freeness

% S.F. - percent of time the project exhibits scale freeness

$k_{min}$  - minimum bound on the power-law distribution

% Dev S.F. - percent of developers described by the power-law distribution

# Dev S.F. - number of developers described by the power-law distribution

$\mu_{cc}$  - mean value of clustering coefficient for latest development cycle

$\sigma_{cc}^2$  - variance of clustering coefficient for latest development cycle

$\beta_{1_{early}}$  - slope parameter for an early development cycle

$\beta_{1_{late}}$  - slope parameter for the most recent development cycle

#### 4.5 Modularity

The clustering coefficient is a means to measure the extent to which developers form cohesive groups. In Figure 5, we present the developer network evolution with respect to clustering coefficients for all subject projects. The evolution of each project is illustrated by a time series that represents the mean clustering coefficient with a light gray boundary to indicate the 99.5% confidence interval. There is one evolutionary profile, in particular, that describes the majority of the subject projects. This profile is characterized by a positive trend component that smoothly converges to a clustering coefficient of 0.45–0.55. For the projects that do not fit this profile, the positive trend components is not observable, possibly because we do not have a complete project history. For example, the development of the Linux kernel was started in 1996, but the publicly available Git repository only has commits dating back until early 2005. In Table 2, we present the results of the mean clustering coefficient (column “ $\mu_{cc}$ ”) and variance (column “ $\sigma_{cc}^2$ ”) for the most recent revision of each project.

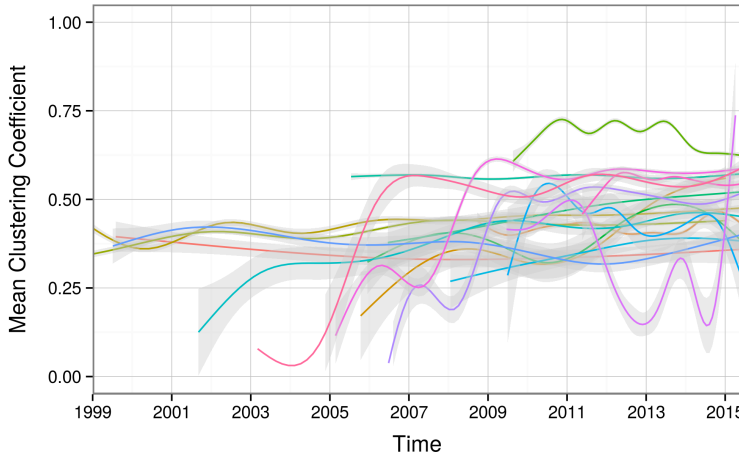


Fig. 5: Clustering-coefficient time series for all subject projects. The light gray boundary indicates the 99.5% confidence interval.

The only project which does not closely adhere to the general pattern of convergence is Qt. The exceptional behavior seen in Qt is likely a consequence of the significant decrease in the number of active developers and possibly represents an evolutionary anti-pattern. The number of developers contributing to Qt is high until 2011 (see Figure 6), but this period is followed by several years of rapid decline. Similarly, we see that the mean clustering coefficient profile fits the general pattern until 2011, where the value suddenly drops and then oscillates before a radical upswing. It is worth noting that Qt is the only subject project exhibiting this pattern, and it is similarly the only subject project that has had such significant decline in number of contributing developers.

For the majority of the projects, the fact that they do not ever significantly exceed a clustering coefficient of 0.55 suggests that there is a limitation to the distribution of coordination requirements in the local developer neighborhood. We observe that there is a preference to achieve a state where roughly half of every developer’s neighbors also have a coordination requirement (i.e. an edge). The evolutionary profiles indicate that developer

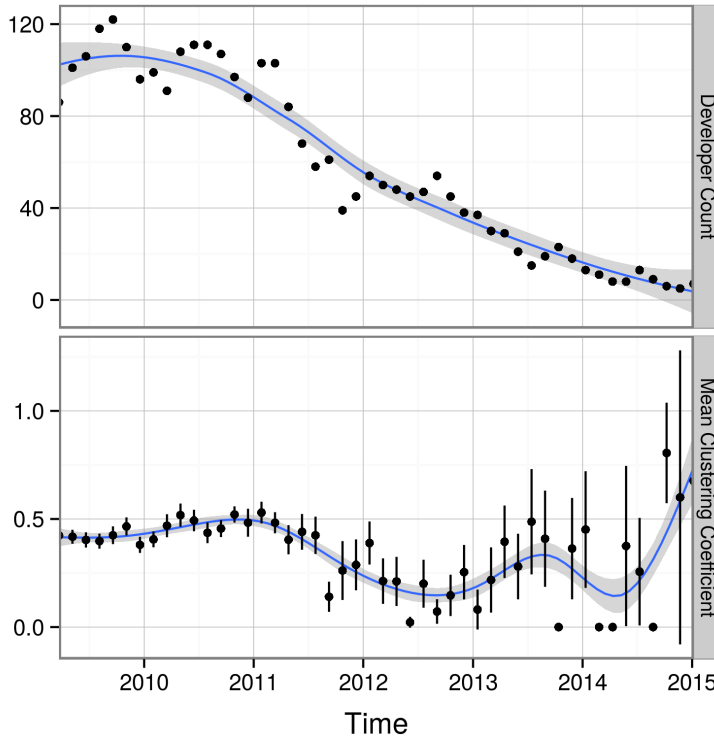


Fig. 6: Developer count and mean clustering coefficient evolution for Qt. The significant decline in developer count co-occurs with instability in the clustering coefficient, indicating that departing developers have a significant impact on the local connectivity of the developer network.

networks evolve according to a process that promotes coordination requirements to increase up to a maximum but prevents the formation of coordination requirements between too many developers. There appears to be no lower bound on the clustering coefficient, but in all cases, an initially low clustering coefficient does tend to converge towards a clustering coefficient of 0.5.

To better understand the mechanism behind the tendency for developers to form cohesive groups, we examined the relationship between network size (i.e., number of developers) and clustering coefficient. In all cases, we found that the clustering coefficient increases with the network size, however, this dependency decreases as the network size increases. This relationship is shown for subject project Django in Figure 7, which illustrates a roughly logarithmic relationship between network size and clustering coefficient. This is a notable result because, in the random graph described in Section 3.4, the clustering coefficient decreases with network size, and in many real-world networks, clustering coefficient and network size are independent (Albert and Barabási, 2002). From this observation, we conclude that developer networks form groups according to a non-random organizational principle that is also different from the preferential-attachment model used to explain many real-world scale-free networks.



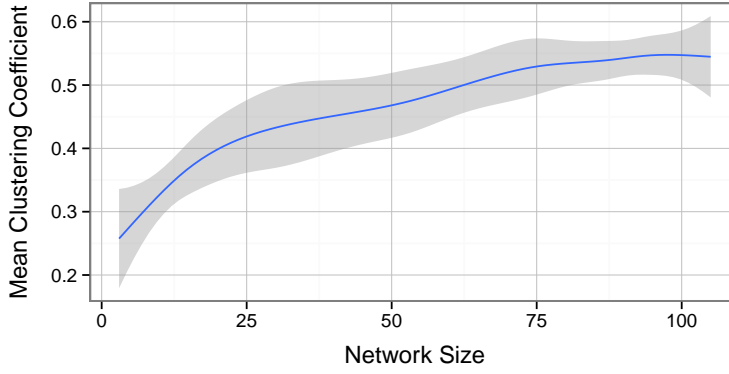


Fig. 7: Clustering coefficient versus network size for the history of Django, with a light gray boundary to indicate the 99.5% confidence intervals.

It has been hypothesized that at a critical upper bound the cost incurred from the overhead of coordination exceeds the benefit of coordinating (Brooks, 1995). Based on the results of our study, the evidence suggests that this bound does exist and developer coordination is constrained to evolve in a manner that promotes groups to form but not to exceed an upper bound. Thus, we *accept H3*.

#### 4.6 Hierarchy

In Section 2.5, we introduced the concept of hierarchy in terms of its relation to scale freeness and modularity: hierarchy is mathematically defined by a linear dependence between the log-transformed clustering coefficient and the node degree. We illustrate the results of applying the method described in Section 2.5 in Figure 8, where the evolution of hierarchy in an early and late state is shown for Firefox. In the early state, we are able to see the network exhibits global hierarchy, because a linear model of the form  $Y = \beta_0 + \beta_1 X$  describes the observed data. The figure indicates that the model is a good fit with an  $R^2$  value of 0.894. Furthermore, the  $p$  value indicates that the linear model slope parameter  $\beta_1$  is significantly different from zero, and so we can conclude that global hierarchy is present. In the late stage, the linear model no longer describes the global set of developers, instead it only describes the high degree nodes. In this case, we can conclude that a global hierarchy is no longer present and hierarchy predominantly exists in the high degree core developer group.

The principal evolutionary trend with respect to hierarchy is the following: In early stages, developers are arranged in a global hierarchy. In later stages, a hybrid structure emerges, where only the core developers are hierarchically arranged, but the global hierarchy is no longer present. We observed that there is a smooth transition between the early and late stages shown for Firefox in Figure 8 that leads to a gradual deconstruction of the global hierarchy. The gradual deconstruction process is shown in Figure 9, where we illustrate the continuous evolution of hierarchy over the entire history of LLVM. Each sample represents the slope parameter  $\beta_1$  of the linear model describing the hierarchy in the project at a single point in time. We see that hierarchy is most significant (largest negative slope) at the start and is progressively lost until virtually no global hierarchy is present (i.e., near zero slope) in the most recent revision. The results for all projects is shown in Table 2, where column “ $\beta_{1_{early}}$ ”

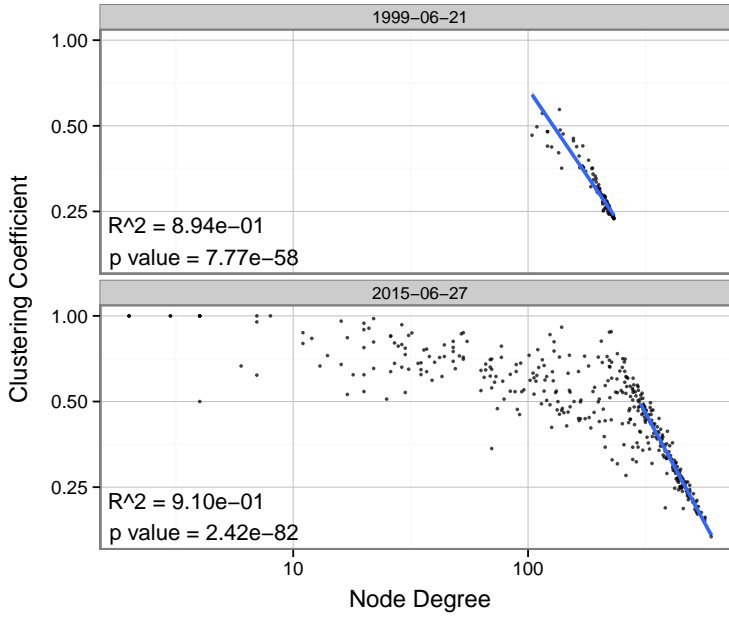


Fig. 8: Early and late stage hierarchy of Firefox. The fitted linear model is superimposed on a scatter plot of degree versus clustering coefficient. In the early stage (top), the linear model describes the complete data set indicating global hierarchy. In the late stage (bottom), global hierarchy is not present since only the high degree nodes are described by the linear model.

represents the linear model slope parameter at an early stage and column “ $\beta_{1_{late}}$ ” represents the slope parameter at a late stage. We are able to see that in all projects except one (Qt 4),  $\beta_{1_{early}} < \beta_{1_{late}}$  indicating that the hierarchy has diminished over time.

The results certainly suggest that, from a global perspective, developer hierarchy diminishes with time, but the mechanism responsible for the transformation is not obvious. To investigate this process further, we examined the high degree nodes (i.e., core developers) and found that they are hierarchically arranged at all times. Furthermore, the mechanism for decomposing the global hierarchy is established through the introduction of low-degree and mid-degree nodes that do not obey the hierarchy established by the high degree nodes. In essence, the developers become divided into two high-level organizational structures: The highest-degree nodes (core developers) are hierarchically arranged and the mid-to-low-degree nodes (peripheral developers) are not hierarchically arranged. This is visible in the late stage scatter plot of Figure 8, where beyond the break point (at a degree of roughly 250), the nodes obey a linear dependence and are thus hierarchically arranged. This evidence suggests that the differences between core and peripheral developers are not entirely explained by their distinct participation levels, but they are also distinct in how they are structurally embedded in the organization.

In summary, we *accept H4*, because the hypothesis is a statement regarding the global structure and, in that sense, the evidence indicates that global hierarchy vanishes over time.

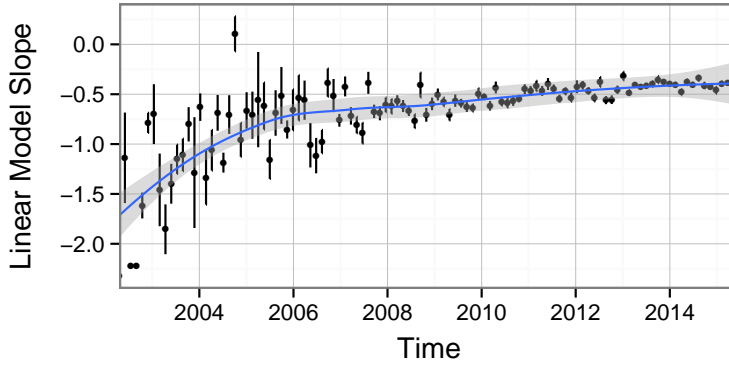


Fig. 9: Evolution of hierarchy for entire history of LLVM. The light gray boundary indicates the 99% confidence interval and error bars indicate the standard error on the slope estimate. The trend indicates that hierarchy is decreasing over time as the linear model slope  $\beta_1$  tends towards zero.

## 5 Threats To Validity

In our study, we draw our conclusions from a manual selection of 18 OSS projects. The manual selection and the choice to analyze only OSS projects is a threat to external validity. We mitigated the consequences by choosing a wide variety of projects that differ in many dimensions and constitute a diverse population. Furthermore, we considered the entire history to prevent temporally biasing our results. We specifically chose only large projects with very active histories, however, our contributions are focused on understanding complexity in developer coordination, and in small projects (e.g., less than 10 developers), the coordination challenges are less severe.

We examined the evolution of developer networks over time, however, it is conceivable that factors other than time have an influence on the observed trends, threatening internal validity. By considering the influence of network size, we accounted for the most likely confounding factor. Furthermore, we found that the trends are often consistent across several projects, and we rigorously employed statistical methods to avoid drawing conclusions from insignificant fluctuations in the data.

Our methodology relies, to some extent, on the integrity of the data in the version-control system to generate a valid developer network, threatening construct validity. Since the version-control system is a critical element of the software-engineering process, it is unlikely that the data would be significantly corrupt. In terms of network construction, the heuristics we rely on have been shown to generate authentic developer networks, but do omit some edges (Joblin et al, 2015). However, a few omitted edges would not severely impact the conclusions of our study. Furthermore, our enhancement of this form of developer networks to recover omitted edges is based on a technique that has been shown to authentically represent the system coupling (Bavota et al, 2013). The operationalizations of scale freeness, modularity, and hierarchy are thoroughly studied and well-established concepts from network analysis. We further relied on the degree of nodes to operationalize the concept core and peripheral developers. Although the application concepts from social network analysis to socio-technical developer networks is relatively new, empirical evidence is accumulating that suggests the metrics are reliable and valid (Meneely and Williams, 2011; Joblin et al, 2015).

## 6 Discussion & Perspectives

The results of our study on the evolution of developer networks revealed several intriguing patterns. We will now discuss the relevance and potential explanation for these network patterns by linking them to software-engineering principles. Specifically, we discuss a likely model for growth of a project, a source of pressure for developers to become more coordinated with time, and the benefits of a hybrid organizational structure that is hierarchical for core developers and non-hierarchical for peripheral developers.

To better understand the growth behavior of developer networks, we examined the relationship between a project's growth state (increasing or decreasing) and the scale-freeness property of its developer network. The model of preferential attachment, which is the predominant generative model for scale-free networks, has the simultaneous requirements that the network must grow and that new nodes have a preference to attach to already well-connected nodes (Barabási and Albert, 1999). We found that, in this regard, the evolution of developer networks into a scale-free state is consistent with the model for preferential attachment. For several projects, the scale-freeness property is only observable during network growth and is lost during periods of growth stagnation or decrease, shown for Node.js in Figure 3. Furthermore, the loss of the scale-freeness property often precedes the stagnation or loss of developers. While it would be premature to make any strong statement about causality, the combination of correlation and preceding in time makes the loss of the scale-free state a conceivable predictor for the loss of growth in the project.

In Section 4.5, we noted that an increasing clustering coefficient is a common evolutionary trend. This is a curious result because, in the ER random graph model (see Section 2.3), the clustering coefficient is independent of the network size, while in the preferential-attachment model, the clustering coefficient decreases with an increasing network size (Ravasz and Barabási, 2003). So, this result begs the question of what the driving force behind this unique evolutionary trend is. From the theory of software evolution, we expect that the natural tendency for an architecture is to become more strongly coupled over time, as complexity increases and initially clean abstraction layers deteriorate (Lehman and Ramil, 2001). Additionally, Conway's law suggests that the organizational structure and the structure of technical artifacts produced by the organization are constrained to mirror each other (Conway, 1968). Based on these principles, we hypothesize that the evolution of the artifact structure is the driving force that influences developers to become more coordinated.

One of the most intriguing characteristics of OSS projects is the strongly inhomogeneous distribution of effort between core and peripheral contributors (Koch, 2004; Toral et al, 2010; Mockus et al, 2000). This characteristic is distinct from typical commercial development processes and is conceivably responsible for enabling OSS projects to scale without reducing overall productivity, which violates conventional software-engineering wisdom (Koch, 2004). Typically, core and peripheral developers are classified based on the number of commits, lines of code, or e-mails they contributed. Interestingly, we discovered that the differences between the two groups are also observable in their organizational structure and stability, where the group of core developers is both hierarchically organized and relatively stable, but the group of peripheral developers is both unstable and not hierarchically organized. We think that the reason for peripheral developers not assimilating into the hierarchy stems from pressures to form a hybrid organizational structure that promotes regularity while also remaining flexible. The process of software development demands a high degree of consistency and, for this reason, hierarchies are appropriate organizational structures. However, hierarchies are intrinsically inflexible structures (Kotter, 2014). In OSS projects, there is pressure for the organizational structure to remain flexible because, as we have shown, OSS projects have

high developer turnover rates for the peripheral developers, who constitute the majority of the contributors. It is conceivable that the existence of a hybrid organizational structure is even a signal of project health by indicating that the organization has responded to the adaptation pressures that are present in OSS development.

## 7 Related Work

Lopez et al. first studied developer coordination by linking developers based on mutual contributions to modules, for a static snapshot of three OSS projects, and concluded that developer networks are not scale free, based on a visual inspection of the cumulative degree distribution (López et al, 2006). Jermakovics et al. constructed networks based on contributions to files for three projects, and they developed a graph-visualization technique to represent the developer organizational structure (Jermakovics et al, 2011). Toral et al. constructed developer communication networks based on the Linux kernel e-mail archives between 2001 and 2006 (Toral et al, 2010). They found that participation inequality is present in the communication network, and they introduced a core developer and peripheral-developer classification scheme. We differentiate our work by analyzing the entire project history and viewing developer coordination as an evolutionary process. Additionally, we use a fully automated and statistically rigorous framework to reduce subjectivity, and we draw our conclusions from 18 projects instead of just two or three. We build on prior work by explaining the commonly observed network features (e.g., participation inequality) in terms of the important structural concepts of scale freeness, modularity, and hierarchy.

Louridas et al. studied structural dependencies between classes and packages of 9 software systems using static source-code analysis techniques (Louridas et al, 2008). They found that power-law distributions are a ubiquitous phenomenon in the dependency structure by fitting a line to the log-scale degree distribution. Our work is complementary by identifying power-law distributions in developers' coordination requirements. This is a step towards an empirical validation of Conway's law by showing that a necessary condition is met regarding the match between the organizational structure and technical artifact structure.

While there are a number of theories regarding turnover and its effects, empirical studies are limited. Foucault et al. examined the relationship between internal and external turnover on software quality in terms of bug density (Foucault et al, 2015). Consistent with current theories, they found that high external turnover has a negative influence on module-level bug density. Others have explored factors that contribute to developer turnover and motivations for long term involvement (Yu et al, 2012; Hynninen et al, 2010; Schilling et al, 2012). Mockus found that developers leaving the projects negatively influence code quality, while new developers entering the project have no influence (Mockus, 2010). Oddly, the results of Mockus and Foucault et al. do not agree, which may suggest that the influence of turnover is dependent on additional context factors. In our work, we primarily focused on the relationship between the turnover characteristics of core and peripheral developer groups and how these distinct groups are structurally embedded in the organization. We then used the distinct turnover rates to rationalize the evolution of the developer network as an optimization process.

Godfrey et al. were the first to study software evolution in OSS and found that the Linux kernel violates principles of software evolution by achieving super-linear growth at the system level (Godfrey and Tu, 2000). This was later supported by evidence extracted from the version control system of 8621 projects on SourceForge.net (Koch, 2004). Koch found that large OSS projects violate several laws of software evolution established for commercial projects (Koch,

2004). Specifically, the study showed that developer productivity is independent of the number of developers in the project—a direct violation of Brooks’ law (Brooks, 1995). Furthermore, participation inequality is common and increases with the system size—a result that we confirmed—but the increase in inequality does not influence developer productivity. Koch proposed that strict modularization, self-organization, and highly decentralized work are responsible for the high efficiency seen in OSS projects, but this was never verified (Koch, 2004). In our study, we found that our more detailed methodology, which considers source-code structure and software coupling, supports prior observations. Furthermore, we were able to extend the body of knowledge by directly studying the evolution of coordination structures that are conjectured to be responsible for the remarkable properties of OSS projects.

## 8 Conclusion

From an organizational perspective, OSS projects are an extreme example of large-scale globally-distributed software engineering, and as such, represent a unique opportunity to study developer-coordination mechanisms. Despite the lack of mandated organizational structures, we found that OSS projects are constrained to evolve according to non-random organizational principles.

Extracting and processing the operational data stored in the version control system proved to be challenging. By pairing our network construction procedure with a sliding-window technique, we were able to identify important insights that would be hidden in a temporally static view. In addition, we enhanced the developer networks by using information retrieval approaches to gain a more rich view of the coordination requirements that exist between developers. We see these technical contributions as meaningful steps towards advancing the techniques for mining software repositories.

Based on our study of 18 OSS projects, we found that, in projects exceeding 50 developers, the coordination structure becomes scale free. We also found that developers tend to have an increasing number of coordination requirements with other developers, but the increasing trend is limited by an upper bound, where coordination requirements exists between roughly half of every developers neighbors. Additionally, we discovered that developers are hierarchically arranged in the early phases of a project, but in later phases the global hierarchy vanishes and a hybrid structure emerges, where core developers exist in the hierarchy and peripheral developers exist outside the hierarchy. With this result, we demonstrated that core and peripheral developers, which are traditionally defined based on their level of participation, also differ in how they are structurally embedded in the project’s coordination structure. Overall, the adaptations that we observed in the structural features balance the opposing constraints of supporting effective coordination and achieving robustness to developer withdrawal. Finally, we discussed how these structural features enable the project in benefiting from a large, but volatile, peripheral developer group, while at the same time, supporting effective coordination and regularity between the much more stable core developer group.

Apart from the general patterns that explain the majority of subject projects, we also noted a number of interesting deviations from the general patterns. For example, during a period of time when Firefox was experiencing notable project delays and turmoil within the developer community, we observed that the scale-freeness property suddenly disappears. In Node.js, there was an oscillatory behavior to the number of contributing developers and the scale-freeness property was lost whenever the project was not in a growing state. Finally, for the few projects which never became scale free, or only for a brief time, a developer group larger than 60 was never sustainable and the number of contributing developers decreased

shortly after reaching a maximum, as was the case for PHP, jQuery, and Apache HTTP. It was often the case that projects deviating significantly from the general patterns were experiencing other negative project conditions such as significant loss in the number of active developers.

Since large-scale studies of software evolution are rare, and studies on the coordination structure are even more rare yet, we see this work as an important step towards understanding the coordination mechanisms that are present in large-scale globally-distributed software engineering. We hope that, by making our analysis infrastructure publicly available, we lower the barrier to contributing to this field of research and accelerate the pace of progress.

## Appendices

### A

#### Function-level semantic coupling

To determine the function-level semantic coupling, we first extracted the implementation for each function in the system, including all source code and comments. We then employed well-established text-mining preprocessing operations with minor modifications for our specific domain requirements. In this framework, each function is treated as a “document” in the text-mining sense of the word, and then the document collection was processed use the following processing pipeline.

*Preprocessing* The preprocessing stage primarily focuses on reducing word diversity and elimination of words that contain little information. *Stemming* is used to reduce words to their root form by removing suffixes (e.g., “ing”, “ly”, “er”, etc.) from each word in the document. Stemming is necessary because, even though a root word may have several forms by adding suffixes, it typically refers to a relatively similar concept in all forms. In software engineering, there is a number of variable-naming conventions, such as letter-case separated (e.g., CamelCase) or delimiter separated words that need to be tokenized appropriately. We added additional preprocessing stages to specifically to handle proper tokenization of popular naming conventions. For example, the variable name “get\_user” or “getUser” are separated into the two words “get” and “user”. One simple example of why this is important is that getters and setters interacting with the same attribute would be incorrectly understood as distinct concepts without appreciating the variable-naming conventions. The final stage of the preprocessing is to remove words which are known to not contain useful information based on a-priori knowledge of the language. For example, words such as “the” are not helpful in determining the domain concept of a document. Removing these words is beneficial for the computational complexity and results by reducing the problem’s dimensionality and attenuating noise in the data.

*Term Weighting* After the preprocessing stage, we arrange all remaining data into a term–document matrix, for mathematical convenience. A term–document matrix is an  $M \times N$  matrix with rows representing terms and columns representing documents. For example, an element of the term document matrix  $TD_{i,j}$  is non-zero when document  $d_j$  contains term  $t_i$ . All elements of the term document matrix are integer weights that indicate the frequency of occurrence of a given term in a given document. We then apply a weight transformation to the term–document matrix based on the statistics of occurrence for each term. Intuition suggests that not all terms in a document are equally important with regard to identifying the domain

concept. The goal of the weighting transformation is to increase the influence of terms that help to identify distinct concepts and decrease the influence of the remaining terms. The particular weighting scheme we applied is called *term frequency-inverse document frequency*:

$$tf-idf_{t,d} = tf_t \times \log \frac{N}{df_t}. \quad (6)$$

The term  $tf_t$  represents the global term frequency across all documents. The second term is the logarithm of the inverse document frequency, where  $N$  is the number of documents in the total collection and  $df_t$  is the number of documents that term  $t$  appears. Upon closer inspection, one can recognize that Equation 6 is: (a) greatest when a term is very frequent, but only appears in a small number of documents, (b) lowest when a term is present in all documents, and (c) between these two extreme cases when a term is infrequent in one document or occurs in many documents.

*Latent Semantic Indexing* Even for a modest-sized software project, the number of terms used in the implementation vocabulary easily exceeds the thousands. The problem with this becomes evident when adopting the vector-space model, where we consider a document as a vector that exists in a space spanned by the terms that comprise the document collection. Fortunately, this very high dimensional space is extremely sparse, which allows us to project the documents into a lower dimensional subspace which makes the semantic similarity computation tractable. We achieve this using a matrix decomposition technique that relies on the singular value decomposition called *latent semantic indexing*. An added benefit of this technique is that it is capable of correctly resolving the relationships of synonymy and polysemy in natural language (Baeza-Yates et al, 1999). Furthermore, LSI has shown evidence to be valid and reliable in the software-engineering domain (Bavota et al, 2013).

*Semantic Similarity* In the final step of the analysis, we determine semantic coupling by computing the similarity between all document vectors projected onto the lower dimensional subspace attained from applying LSI. We operationalize the similarity between two document vectors in the latent space using cosine similarity

$$\text{similarity}(\mathbf{d}_a, \mathbf{d}_b) = \frac{\mathbf{d}_a \cdot \mathbf{d}_b}{\|\mathbf{d}_a\| \|\mathbf{d}_b\|}, \quad (7)$$

where the numerator is the dot product between the two document vectors and the denominator is the multiplication of the magnitude of the two document vectors. Intuitively, cosine similarity expresses the difference in the angle between the two document vectors; it equals 1, when the two vectors are parallel, and 0, if they are orthogonal. Two source-code artifacts are then considered to be semantically coupled if the cosine similarity exceeds a given threshold. We experimented extensively with a number of thresholds by manually inspecting the results and judging whether the functions were, in fact, semantically related using architectural knowledge of a well-known project. We found that a threshold of 0.5 was able to identify most semantic relationships with only a very small number of false positives. We did however cautiously chose the threshold to optimize to avoid false positives rather than false negatives.

## References

Albert R, Barabási AL (2002) Statistical mechanics of complex networks. Reviews of modern physics 74(1):47



- Arias TBC, van der Spek P, Avgeriou P (2011) A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering* 16(5):544–586
- Arnold RS, Bohner SA (1993) Impact analysis-towards a framework for comparison. In: *Proc. International Conference on Software Maintenance*, IEEE, pp 292–301
- Atkinson AB (1970) On the measurement of inequality. *Journal of Economic Theory* 2(3):244–263
- Baeza-Yates R, Ribeiro-Neto B, et al (1999) *Modern Information Retrieval*. Addison-Wesley
- Barabási AL, Albert R (1999) Emergence of scaling in random networks. *Science* 286(5439):509–512
- Barabási AL, Jeong H, Néda Z, Ravasz E, Schubert A, Vicsek T (2002) Evolution of the social network of scientific collaborations. *Physica A: Statistical mechanics and its applications* 311(3):590–614
- Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2013) An empirical study on the developers' perception of software coupling. In: *Proc. International Conference on Software Engineering*, IEEE, pp 692–701
- Begel A, Khoo YP, Zimmermann T (2010) Codebook: Discovering and exploiting relationships in software repositories. In: *Proc. International Conference on Software Engineering*, ACM, pp 125–134
- Bernard HR, Killworth PD, Evans MJ, McCarty C, Shelley GA (1988) Studying social relations cross-culturally. *Ethnology* 27(2):155–179
- Bishop CM (2006) *Pattern recognition and machine learning*. Springer
- Boccaletti S, Latora V, Moreno Y, Chavez M, Hwang DU (2006) Complex networks: Structure and dynamics. *Physics reports* 424(4):175–308
- Boehm BW (ed) (1989) *Software Risk Management*. IEEE
- Brooks FP (1995) *The mythical man-month*. Addison-Wesley Reading
- Cataldo M, Herbsleb JD (2013) Coordination breakdowns and their impact on development productivity and Software Failures. *IEEE Transactions on Software Engineering* 39(3):343–360
- Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM (2006) Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In: *Proc. Conference on Computer Supported Cooperative Work*, ACM, pp 353–362
- Cataldo M, Herbsleb JD, Carley KM (2008) Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In: *Proc. International Symposium on Empirical Software Engineering and Measurement*, ACM, pp 2–11
- Cataldo M, Mockus A, Roberts JA, Herbsleb JD (2009) Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35(6):864–878
- Clauset A, Shalizi CR, Newman MEJ (2009) Power-law distributions in empirical data. *SIAM Review* 51(4):661–703
- Conway ME (1968) How do committees invent. *Datamation* 14(4):28–31
- Crowston K, Howison J (2005) The social structure of free and open source software development. *First Monday* 10(2)
- Crowston K, Kangning W, Howison J, Wiggins A (2012) Free/libre open-source software development: What we know and what we do not know. *ACM computing surveys* 44(2)
- DiBona C, Ockman S, Stone M (eds) (1999) *Open sources: Voices from the open source revolution*. O'Reilly Media & Associates, Inc.
- Dorogovtsev SN, Mendes JF (2013) *Evolution of networks: From biological nets to the Internet and WWW*. Oxford University Press

- Erdős P, Rényi A (1959) On random graphs. *Publicationes Mathematicae* 6:290–297
- Espinosa JA, Slaughter SA, Kraut RE, Herbsleb JD (2007) Familiarity, complexity, and team performance in geographically distributed software development. *Organization Science* 18(4):613–630
- Foucault M, Palyart M, Blanc X, Murphy GC, Falleri JR (2015) Impact of developer turnover on quality in open-source software. In: *Proc. International Symposium on Foundations of Software Engineering*, ACM
- Godfrey MW, Tu Q (2000) Evolution in open source software: A case study. In: *Proc. International Conference on Software Maintenance*, IEEE, pp 131–
- Goldstein M, Morris S, Yen G (2004) Problems with fitting to the power-law distribution. *European Physical Journal B–Condensed Matter* 41(2)
- Hinds P, McGrath C (2006) Structures that work: Social structure, work structure and coordination ease in geographically distributed teams. In: *Proc. Conference on Computer Supported Cooperative Work*, ACM, pp 343–352
- Huang NE, Shen Z, Long SR, Wu MC, Shih HH, Zheng Q, Yen NC, Tung CC, Liu HH (1998) The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis. *Proc Royal Society of London A: Mathematical, Physical and Engineering Sciences* 454(1971):903–995
- Huselid MA (1995) The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of Management journal* 38(3):635–672
- Hynninen P, Piri A, Niinimäki T (2010) Off-site commitment and voluntary turnover in gsd projects. In: *Proc. International Conference on Global Software Engineering*, IEEE, pp 145–154
- Jeong H, Tombor B, Albert R, Oltvai ZN, Barabási AL (2000) The large-scale organization of metabolic networks. *Nature* 407(6804):651–654
- Jermakovics A, Sillitti A, Succi G (2011) Mining and visualizing developer networks from version control systems. In: *Proc. International Workshop on Cooperative and Human Aspects of Software Engineering*, ACM, pp 24–31
- Joblin M, Mauerer W, Apel S, Siegmund J, Riehle D (2015) From developer networks to verified communities: A fine-grained approach. In: *Proc. International Conference on Software Engineering*, IEEE, pp 563–573
- Koch S (2004) Profiling an open source project ecology and its programmers. *Electronic Markets* 14(2):77–88
- Kotter JP (2014) *Accelerate: Building Strategic Agility for a Faster-moving World*. Harvard Business Review Press
- Lehman MM, Ramil JF (2001) Rules and tools for software evolution planning and management. *Annals of Software Engineering* 11(1):15–44
- Lehman MM, Ramil JF, Wernick PD, Perry DE, M W (1997) Metrics and laws of software evolution the nineties view. In: *Proc. Software Metrics Symposium*, IEEE, pp 20–32
- López L, Robles G, Jesús, Herraiz I (2006) Applying social network analysis techniques to community-driven libre software projects. *International Journal of Information Technology and Web Engineering* 1(3):27–48
- Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. *ACM Transactions on Software Engineering and Methodology* 18(1):2
- Martinez-Romo J, Robles G, Gonzalez-Barahona JM, Ortuño-Perez M (2008) Using social network analysis techniques to study collaboration between a floss community and a company. In: *Open Source Development, Communities and Quality*, Springer, pp 171–186

- 
- Mauerer W, Jaeger MC (2013) Open source engineering processes. *Information Technology* 55(5):196–203
- Meneely A, Williams L (2011) Socio-technical developer networks: Should we trust our measurements? In: *Proc. International Conference on Software Engineering, ACM*, pp 281–290
- Meneely A, Williams L, Snipes W, Osborne J (2008) Predicting failures with developer networks and social network analysis. In: *Proc. Foundations of Software Engineering, ACM*, pp 13–23
- Mockus A (2010) Organizational volatility and its effects on software defects. In: *Proc. International Symposium on Foundations of Software Engineering, ACM*, pp 117–126
- Mockus A, Fielding RT, Herbsleb J (2000) A case study of open source software development: The Apache server. In: *Proc. International Conference on Software Engineering*, pp 263–272
- Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and Mozilla. *ACM Transactions Software Engineering Methodology* 11(3):309–346
- Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T (2009) Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* 14(1):5–32
- Ravasz E, Barabási AL (2003) Hierarchical organization in complex networks. *Physical Review E* 67(2):026,112
- Robles G, Gonzalez-Barahona J, Herraiz I (2009) Evolution of the core team of developers in libre software projects. In: *Proc. International Working Conference on Mining Software Repositories, IEEE*, pp 167–170
- Schilling A, Laumer S, Weitzel T (2012) Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in floss projects. In: *Proc. International Conference on System Sciences, IEEE Computer Society, HICSS '12*
- Sosa ME, Eppinger SD, Rowles CM (2004) The misalignment of product architecture and organizational structure in complex product development. *Manage Sci* 50(12):1674–1689
- Terceiro A, Rios LR, Chavez C (2010) An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In: *Proc. Brazilian Symposium on Software Engineering, IEEE*, pp 21–29
- Toral S, Martínez-Torres M, Barrero F (2010) Analysis of virtual communities supporting oss projects using social network analysis. *Information and Software Technology* 52(3):296–303
- Yu Y, Benlian A, Hess T (2012) An empirical study of volunteer members' perceived turnover in open source software projects. In: *Proc. International Conference on System Sciences, IEEE*, pp 3396–3405